38 | 总结回顾面向对象、设计原则、编程规范、重构技巧等知识点

2020-01-29 王争

设计模式之美 进入课程 >

讲述：冯永吉

时长 34:06 大小 27.33M

到今天为止，设计原则和思想已经全部讲完了，其中包括：面向对象、设计原则、规范与重构三个模块的内容。除此之外，我们还学习了贯穿整个专栏的代码质量评判标准。专栏的进度已经接近一半，马上就要进入设计模式内容的学习了。在此之前，我先带你一块来总结回顾一下，我们已经学过的所有知识点。

今天的内容比较多，有一万多字，但都是之前学过的，算是一个总结回顾，主要是想带你复习一下，温故而知新。如果你看了之后，感觉都有印象，那就说明学得还不错；如果还能在脑子里形成自己的知识架构，闭上眼睛都能回忆上来，那说明你学得很好；如果能有[illegible]理解，并且在项目开发中，开始思考代码质量问题，开始用已经学过的设计原则和思想来审视代码，那说明你已经掌握这些内容的精髓。

一、代码质量评判标准

如何评价代码质量的高低？

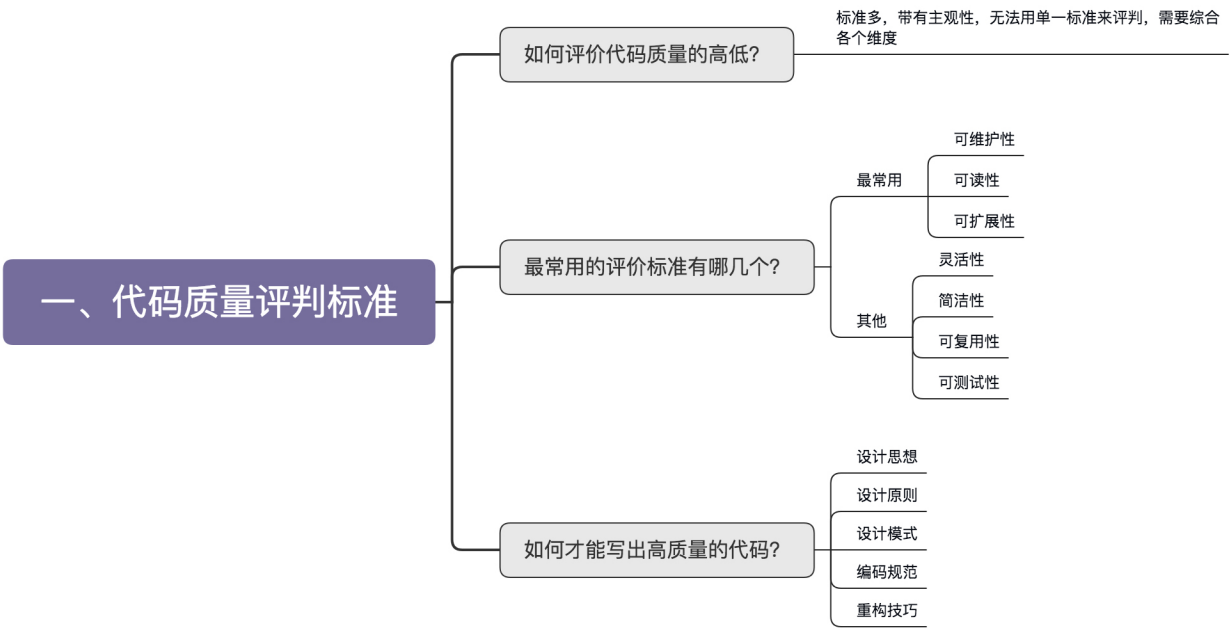
代码质量的评价有很强的主观性，描述代码质量的词汇也有很多，比如可读性、可维护性、灵活、优雅、简洁。这些词汇是从不同的维度去评价代码质量的。它们之间有互相作用，并不是独立的，比如，代码的可读性好、可扩展性好就意味着代码的可维护性好。代码质量高低是一个综合各种因素得到的结论。我们并不能通过单一维度去评价一段代码的好坏。

最常用的评价标准有哪几个？

最常用到几个评判代码质量的标准有：可维护性、可读性、可扩展性、灵活性、简洁性、可复用性、可测试性。其中，可维护性、可读性、可扩展性又是提到最多的、最重要的三个评价标准。

如何才能写出高质量的代码？

要写出高质量代码，我们就需要掌握一些更加细化、更加能落地的编程方法论，这就包含面向对象设计思想、设计原则、设计模式、编码规范、重构技巧等。



极客时间

二、面向对象

1. 面向对象概述

现在，主流的编程范式或者编程风格有三种，它们分别是面向过程、面向对象和函数式编程。面向对象这种编程风格又是这其中最主流的。现在比较流行的编程语言大部分都是面向对象编程语言。大部分项目也都是基于面向对象编程风格开发的。面向对象编程因为其具有丰富的特性（封装、抽象、继承、多态），可以实现很多复杂的设计思路，是很多设计原则、设计模式编码实现的基础。

2. 面向对象四大特性

封装也叫作信息隐藏或者数据访问保护。类通过暴露有限的访问接口，授权外部仅能通过类提供的方法来访问内部信息或者数据。它需要编程语言提供权限访问控制语法来支持，例如 Java 中的 private、protected、public 关键字。封装特性存在的意义，一方面是保护数据不被随意修改，提高代码的可维护性；另一方面是仅暴露有限的必要接口，提高类的易用性。

如果说封装主要讲如何隐藏信息、保护数据，那抽象就是讲如何隐藏方法的具体实现，让使用者只需要关心方法提供了哪些功能，不需要知道这些功能是如何实现的。抽象可以通过接口类或者抽象类来实现。抽象存在的意义，一方面是修改实现不需要改变定义；另一方面，它也是处理复杂系统的有效手段，能有效地过滤掉不必要关注的信息。

继承用来表示类之间的 is-a 关系，分为两种模式：单继承和多继承。单继承表示一个子类只继承一个父类，多继承表示一个子类可以继承多个父类。为了实现继承这个特性，编程语言需要提供特殊的语法机制来支持。继承主要是用来解决代码复用的问题。

多态是指子类可以替换父类，在实际的代码运行过程中，调用子类的方法实现。多态这种特性也需要编程语言提供特殊的语法机制来实现，比如继承、接口类、duck-typing。多态可以提高代码的扩展性和复用性，是很多设计模式、设计原则、编程技巧的代码实现基础。

3. 面向对象 VS 面向过程

面向对象编程相比面向过程编程的优势主要有三个。

对于大规模复杂程序的开发，程序的处理流程并非单一的一条主线，而是错综复杂的网状结构。面向对象编程比起面向过程编程，更能应对这种复杂类型的程序开发。

面向对象编程相比面向过程编程，具有更加丰富的特性（封装、抽象、继承、多态）。利用这些特性编写出来的代码，更加易扩展、易复用、易维护。

从编程语言跟机器打交道方式的演进规律中，我们可以总结出：面向对象编程语言比起面向过程编程语言，更加人性化、更加高级、更加智能。

面向对象编程一般使用面向对象编程语言来进行，但是，不用面向对象编程语言，我们照样可以进行面向对象编程。反过来讲，即便我们使用面向对象编程语言，写出来的代码也不一定是面向对象编程风格的，也有可能是面向过程编程风格的。

面向对象和面向过程两种编程风格并不是非黑即白、完全对立的。在用面向对象编程语言开发的软件中，面向过程风格的代码并不少见，甚至在一些标准的开发库（比如 JDK、Apache Commons、Google Guava）中，也有很多面向过程风格的代码。

不管使用面向过程还是面向对象哪种风格来写代码，我们最终的目的还是写出易维护、易读、易复用、易扩展的高质量代码。只要我们能避免面向过程编程风格的一些弊端，控制好

它的副作用，在掌控范围内为我们所用，我们就大可不用避讳在面向对象编程中写面向过程风格的代码。

4. 面向对象分析、设计与编程

面向对象分析（OOA）、面向对象设计（OOD）、面向对象编程（OOP），是面向对象开发的三个主要环节。简单点讲，面向对象分析就是要搞清楚做什么，面向对象设计就是要搞清楚怎么做，面向对象编程就是将分析和设计的的结果翻译成代码的过程。

需求分析的过程实际上是一个不断迭代优化的过程。我们不要试图一下就给出一个完美的解决方案，而是先给出一个粗糙的、基础的方案，有一个迭代的基础，然后再慢慢优化。这样一个思考过程能让我们摆脱无从下手的窘境。

面向对象设计和实现要做的事情就是把合适的代码放到合适的类中。至于到底选择哪种划分方法，判定的标准是让代码尽量地满足“松耦合、高内聚”、单一职责、对扩展开放对修改关闭等我们之前讲到的各种设计原则和思想，尽量地做到代码可复用、易读、易扩展、易维护。

面向对象分析的产出是详细的需求描述。面向对象设计的产出是类。在面向对象设计这一环节中，我们将需求描述转化为具体的类的设计。这个环节的工作可以拆分为下面四个部分。

划分职责进而识别出有哪些类

根据需求描述，我们把其中涉及的功能点，一个一个罗列出来，然后再去看哪些功能点职责相近，操作同样的属性，可否归为同一个类。

定义类及其属性和方法

我们识别出需求描述中的动词，作为候选的方法，再进一步过滤筛选出真正的方法，把功能点中涉及的名词，作为候选属性，然后同样再进行过滤筛选。

定义类与类之间的交互关系

UML 统一建模语言中定义了六种类之间的关系。它们分别是：泛化、实现、关联、聚合、组合、依赖。我们从更加贴近编程的角度，对类与类之间的关系做了调整，保留了四个关

系：泛化、实现、组合、依赖。

将类组装起来并提供执行入口

我们要将所有的类组装在一起，提供一个执行入口。这个入口可能是一个 main() 函数，也可能是一组给外部用的 API 接口。通过这个入口，我们能触发整个代码跑起来。

5. 接口 VS 抽象类

抽象类不允许被实例化，只能被继承。它可以包含属性和方法。方法既可以包含代码实现，也可以不包含代码实现。不包含代码实现的方法叫作抽象方法。子类继承抽象类，必须实现抽象类中的所有抽象方法。接口不能包含属性（Java 可以定义静态常量），只能声明方法，方法不能包含代码实现（Java8 以后可以有默认实现）。类实现接口的时候，必须实现接口中声明的所有方法。

抽象类是对成员变量和方法的抽象，是一种 is-a 关系，是为了解决代码复用问题。接口仅仅是对方法的抽象，是一种 has-a 关系，表示具有某一组行为特性，是为了解决解耦问题，隔离接口和具体的实现，提高代码的扩展性。

什么时候该用抽象类？什么时候该用接口？实际上，判断的标准很简单。如果要表示一种 is-a 的关系，并且是为了解决代码复用问题，我们就用抽象类；如果要表示□一种 has-a 关系，并且是为了解决抽象而非代码复用问题，那我们就用接口。

6. 基于接口而非实现编程

应用这条原则，可以将接口和实现相分离，封装不稳定的实现，暴露稳定的接口。上游系统面向接口而非实现编程，不依赖不稳定的实现细节，这样当实现发生变化的时候，上游系统的代码基本上不需要做改动，以此来降低耦合性，提高扩展性。

实际上，“基于接口而非实现编程”这条原则的另一个表述方式是，“基于抽象而非实现编程”。后者的表述方式其实更能体现这条原则的设计初衷。在软件开发中，最大的挑战之一就是需求的不断变化，这也是考验代码设计好坏的一个标准。

越抽象、越顶层、越脱离具体某一实现的设计，越能提高代码的灵活性，越能应对未来的需求变化。好的代码设计，不仅能应对当下的需求，而且在将来需求发生变化的时候，仍然能

够在不破坏原有代码设计的情况下灵活应对。而抽象就是提高代码扩展性、灵活性、可维护性最有效的手段之一。

7. 多用组合少用继承

为什么不推荐使用继承?

继承是面向对象的四大特性之一，用来表示类之间的 is-a 关系，可以解决代码复用的问题。虽然继承有诸多作用，但继承层次过深、过复杂，也会影响到代码的可维护性。在这种情况下，我们应该尽量少用，甚至不用继承。

组合相比继承有哪些优势?

继承主要有三个作用：表示 is-a 关系、支持多态特性、代码复用。而这三个作用都可以通过组合、接口、委托三个技术手段来达成。除此之外，利用组合还能解决层次过深、过复杂的继承关系影响代码可维护性的问题。

如何判断该用组合还是继承?

尽管我们鼓励多用组合少用继承，但组合也并不是完美的，继承也并非一无是处。在实际的项目开发中，我们还是要根据具体的情况，来选择该用继承还是组合。如果类之间的继承结构稳定，层次比较浅，关系不复杂，我们就可以大胆地使用继承。反之，我们就尽量使用组合来替代继承。除此之外，还有一些设计模式、特殊的应用场景，会固定使用继承或者组合。

8. 贫血模型 VS 充血模型

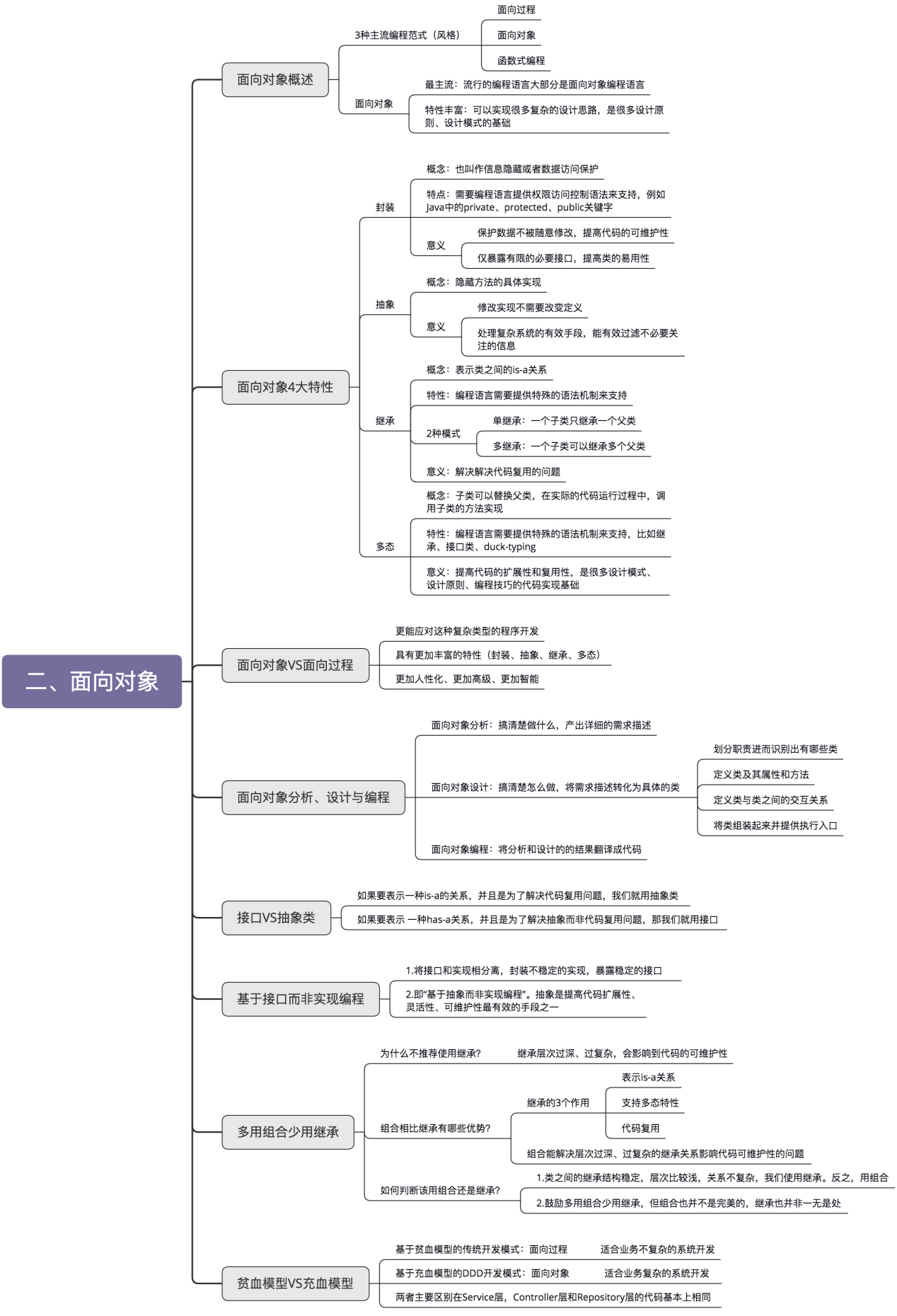
我们平时做 Web 项目的业务开发，大部分都是基于贫血模型的 MVC 三层架构，在专栏中我把它称为传统的开发模式。之所以称之为“传统”，是相对于新兴的基于充血模型的 DDD 开发模式来说的。基于贫血模型的传统开发模式，是典型的面向过程的编程风格。相反，基于充血模型的 DDD 开发模式，是典型的面向对象的编程风格。

不过，DDD 也并非银弹。对于业务不复杂的系统开发来说，基于贫血模型的传统开发模式简单够用，基于充血模型的 DDD 开发模式有点大材小用，无法发挥作用。相反，对于业务复杂的系统开发来说，基于充血模型的 DDD 开发模式，因为前期需要在设计上投入更多时

间和精力，来提高代码的复用性和可维护性，所以相比基于贫血模型的开发模式，更加有优势。

基于充血模型的 DDD 开发模式跟基于贫血模型的传统开发模式相比，主要区别在 Service 层。在基于充血模型的开发模式下，我们将部分原来在 Service 类中的业务逻辑移动到了一个充血的 Domain 领域模型中，让 Service 类的实现依赖这个 Domain 类。不过，Service 类并不会完全移除，而是负责一些不适合放在 Domain 类中的功能。比如，负责与 Repository 层打交道、跨领域模型的业务聚合功能、幂等事务等非功能性的工作。

基于充血模型的 DDD 开发模式跟基于贫血模型的传统开发模式相比，Controller 层和 Repository 层的代码基本上相同。这是因为，Repository 层的 Entity 生命周期有限，Controller 层的 VO 只是单纯作为一种 DTO。两部分的业务逻辑都不会太复杂。业务逻辑主要集中在 Service 层。所以，Repository 层和 Controller 层继续沿用贫血模型的设计思路是没有问题的。


二、面向对象
面向对象概述
3种主流编程范式（风格）
面向过程
面向对象
函数式编程
面向对象
最主流：流行的编程语言大部分是面向对象编程语言
特性丰富：可以实现很多复杂的设计思路，是很多设计原则、设计模式的基础
面向对象4大特性
封装
概念：也叫作信息隐藏或者数据访问保护
特点：需要编程语言提供权限访问控制语法来支持，例如Java中的private、protected、public关键字
意义
保护数据不被随意修改，提高代码的可维护性
仅暴露有限的必要接口，提高类的易用性
抽象
概念：隐藏方法的具体实现
意义
修改实现不需要改变定义
处理复杂系统的有效手段，能有效过滤不必要关注的信息
继承
概念：表示类之间的is-a关系
特性：编程语言需要提供特殊的语法机制来支持
2种模式
单继承：一个子类只继承一个父类
多继承：一个子类可以继承多个父类
意义：解决解决代码复用的问题
多态
概念：子类可以替换父类，在实际的代码运行过程中，调用子类的方法实现
特性：编程语言需要提供特殊的语法机制来支持，比如继承、接口类、duck-typing
意义：提高代码的扩展性和复用性，是很多设计模式、设计原则、编程技巧的代码实现基础
面向对象VS面向过程
更能应对这种复杂类型的程序开发
具有更加丰富的特性（封装、抽象、继承、多态）
更加人性化、更加高级、更加智能
面向对象分析、设计与编程
面向对象分析：搞清楚做什么，产出详细的需求描述
面向对象设计：搞清楚怎么做，将需求描述转化为具体的类
划分职责进而识别出有哪些类
定义类及其属性和方法
定义类与类之间的交互关系
将类组装起来并提供执行入口
面向对象编程：将分析和设计的的结果翻译成代码
接口VS抽象类
如果要表示一种is-a的关系，并且是为了解决代码复用问题，我们就用抽象类
如果要表示 一种has-a关系，并且是为了解决抽象而非代码复用问题，那我们就用接口
基于接口而非实现编程
1.将接口和实现相分离，封装不稳定的实现，暴露稳定的接口
2.即“基于抽象而非实现编程”。抽象是提高代码扩展性、灵活性、可维护性最有效的手段之一
多用组合少用继承
为什么不推荐使用继承?
继承层次过深、过复杂，会影响到代码的可维护性
组合相比继承有哪些优势?
继承的3个作用
表示is-a关系
支持多态特性
代码复用
组合能解决层次过深、过复杂的继承关系影响代码可维护性的问题
如何判断该用组合还是继承?
1.类之间的继承结构稳定，层次比较浅，关系不复杂，我们使用继承。反之，用组合
2.鼓励多用组合少用继承，但组合也并不是完美的，继承也并非一无是处
贫血模型VS充血模型
基于贫血模型的传统开发模式：面向过程
适合业务不复杂的系统开发
基于充血模型的DDD开发模式：面向对象
适合业务复杂的系统开发
两者主要区别在Service层，Controller层和Repository层的代码基本上相同

三、设计原则

1.SOLID 原则：SRP 单一职责原则

一个类只负责完成一个职责或者功能。单一职责原则通过避免设计大而全的类，避免将不相关的功能耦合在一起，来提高类的内聚性。同时，类职责单一，类依赖的和被依赖的其他类也会变少，减少了代码的耦合性，以此来实现代码的高内聚、松耦合。但是，如果拆分得过细，实际上会适得其反，反倒会降低内聚性，也会影响代码的可维护性。

不同的应用场景、不同阶段的需求背景、不同的业务层面，对同一个类的职责是否单一，可能会有不同的判定结果。实际上，一些侧面的判断指标更具有指导意义和可执行性，比如，出现下面这些情况就有可能说明这类的设计不满足单一职责原则：

类中的代码行数、函数或者属性过多；

类依赖的其他类过多或者依赖类的其他类过多；

私有方法过多；

比较难给类起一个合适的名字；

类中大量的方法都是集中操作类中的某几个属性。

2.SOLID 原则：OCP 开闭原则

如何理解“对扩展开放、修改关闭”？

添加一个新的功能，应该是通过在已有代码基础上扩展代码（新增模块、类、方法、属性等），而非修改已有代码（修改模块、类、方法、属性等）的方式来完成。关于定义，我们有两点要注意。第一点是，开闭原则并不是说完全杜绝修改，而是以最小的修改代码的代价来完成新功能的开发。第二点是，同样的代码改动，在粗代码粒度下，可能被认定为“修改”；在细代码粒度下，可能又被认定为“扩展”。

如何做到“对扩展开放、修改关闭”？

我们要时刻具备扩展意识、抽象意识、封装意识。在写代码的时候，我们要多花点时间思考一下，这段代码未来可能有哪些需求变更，如何设计代码结构，事先留好扩展点，以便在未

来需求变更的时候，在不改动代码整体结构、做到最小代码改动的情况下，将新的代码灵活地插入到扩展点上。

很多设计原则、设计思想、设计模式，都是以提高代码的扩展性为最终目的的。特别是 23 种经典设计模式，大部分都是为了解决代码的扩展性问题而总结出来的，都是以开闭原则为指导原则的。最常用来提高代码扩展性的方法有：多态、依赖注入、基于接口而非实现编程，以及大部分的设计模式（比如，装饰、策略、模板、职责链、状态）。

3.SOLID 原则：LSP 里式替换原则

子类对象（object of subtype/derived class）能够替换程序（program）中父类对象（object of base/parent class）出现的任何地方，并且保证原来程序的逻辑行为（behavior）不变及正确性不被破坏。

里式替换原则是用来指导继承关系中子类该如何设计的一个原则。理解里式替换原则，最核心的就是理解“design by contract，按照协议来设计”这几个字。父类定义了函数的“约定”（或者叫协议），那子类可以改变函数的内部实现逻辑，但不能改变函数的原有“约定”。这里的“约定”包括：函数声明要实现的功能；对输入、输出、异常的约定；甚至包括注释中所罗列的任何特殊说明。

理解这个原则，我们还要弄明白，里式替换原则跟多态的区别。虽然从定义描述和代码实现上来看，多态和里式替换有点类似，但它们关注的角度是不一样的。多态是面向对象编程的一大特性，也是面向对象编程语言的一种语法。它是一种代码实现的思路。而里式替换是一种设计原则，用来指导继承关系中子类该如何设计，子类的设计要保证在替换父类的时候，不改变原有程序的逻辑及不破坏原有程序的正确性。

4.SOLID 原则：ISP 接口隔离原则

接口隔离原则的描述是：客户端不应该强迫依赖它不需要的接口。其中的“客户端”，可以理解为接口的调用者或者使用者。理解“接口隔离原则”的重点是理解其中的“接口”二字。这里有三种不同的理解。

如果把“接口”理解为一组接口集合，可以是某个微服务的接口，也可以是某个类库的接口等。如果部分接口只被部分调用者使用，我们就需要将这部分接口隔离出来，单独给这部分调用者使用，而不强迫其他调用者也依赖这部分不会被用到的接口。

如果把“接口”理解为单个 API 接口或函数，部分调用者只需要函数中的部分功能，那我们就需要把函数拆分成粒度更细的多个函数，让调用者只依赖它需要的那个细粒度函数。

如果把“接口”理解为 OOP 中的接口，也可以理解为面向对象编程语言中的接口语法。那接口的设计要尽量单一，不要让接口的实现类和调用者，依赖不需要的接口函数。

单一职责原则针对的是模块、类、接口的设计。接口隔离原则相对于单一职责原则，一方面更侧重于接口的设计，另一方面它的思考的角度也是不同的。接口隔离原则提供了一种判断接口的职责是否单一的标准：通过调用者如何使用接口来间接地判定。如果调用者只使用部分接口或接口的部分功能，那接口的设计就不够职责单一。

5.SOLID 原则：DIP 依赖倒置原则

控制反转：实际上，控制反转是一个比较笼统的设计思想，并不是一种具体的实现方法，一般用来指导框架层面的设计。这里所说的“控制”指的是对程序执行流程的控制，而“反转”指的是在没有使用框架之前，程序员自己控制整个程序的执行。在使用框架之后，整个程序的执行流程通过框架来控制。流程的控制权从程序员“反转”给了框架。

依赖注入：依赖注入和控制反转恰恰相反，它是一种具体的编码技巧。我们不通过 new 的方式在类内部创建依赖类的对象，而是将依赖的类对象在外部创建好之后，通过构造函数、函数参数等方式传递（或“注入”）给类来使用。

依赖注入框架：我们通过依赖注入框架提供的扩展点，简单配置一下所有需要的类及其类与类之间的依赖关系，就可以实现由框架来自动创建对象、管理对象的生命周期、依赖注入等原本需要程序员来做的事情。

依赖反转原则：依赖反转原则也叫作依赖倒置原则。这条原则跟控制反转有点类似，主要用来指导框架层面的设计。高层模块不依赖低层模块，它们共同依赖同一个抽象。抽象不需要依赖具体实现细节，具体实现细节依赖抽象。

6.KISS、YAGNI 原则

KISS 原则的中文描述是：尽量保持简单。KISS 原则是保持代码可读和可维护的重要手段。KISS 原则中的“简单“”并不是以代码行数来考量的。代码行数越少并不代表代码越简单，我们还要考虑逻辑复杂度、实现难度、代码的可读性等。而且，本身就复杂的问题，用

复杂的方法解决，也并不违背 KISS 原则。除此之外，同样的代码，在某个业务场景下满足 KISS 原则，换一个应用场景可能就不满足了。

对于如何写出满足 KISS 原则的代码，我总结了下面几条指导原则：

不要使用同事可能不懂的技术来实现代码；

不要重复造轮子，善于使用已经有的工具类库；

不要过度优化。

YAGNI 原则的英文全称是：You Ain’t Gonna Need It。直译就是：你不会需要它。这条原则也算是万金油了。当用在软件开发中的时候，它的意思是：不要去设计当前用不到的功能；不要去编写当前用不到的代码。实际上，这条原则的核心思想就是：不要做过度设计。

YAGNI 原则跟 KISS 原则并非一回事儿。KISS 原则讲的是“如何做”的问题（尽量保持简单），而 YAGNI 原则说的是“要不要做”的问题（当前不需要的就不要做）。

7.DRY 原则

DRY 原则中文描述是：不要重复自己，将它应用在编程中，可以理解为：不要写重复的代码。

专栏中讲到了三种代码重复的情况：实现逻辑重复、功能语义重复、代码执行重复。实现逻辑重复，但功能语义不重复的代码，并不违反 DRY 原则。实现逻辑不重复，但功能语义重复的代码，也算是违反 DRY 原则。而代码执行重复也算是违反 DRY 原则。

除此之外，我们还讲到了提高代码复用性的一些手段，包括：减少代码耦合、满足单一职责原则、模块化、业务与非业务逻辑分离、通用代码下沉、继承、多态、抽象、封装、应用模板等设计模式。复用意识也非常重要。在设计每个模块、类、函数的时候，要像设计一个外部 API 一样去思考它的复用性。

我们在第一次写代码的时候，如果当下没有复用的需求，而未来的复用需求也不是特别明确，并且开发可复用代码的成本比较高，那我们就不需要考虑代码的复用性。在之后开发新的功能的时候，发现可以复用之前写的这段代码，那我们就重构这段代码，让其变得更加可复用。

相比于代码的可复用性，DRY 原则适用性更强些。我们可以不写可复用的代码，但一定不能写重复的代码。

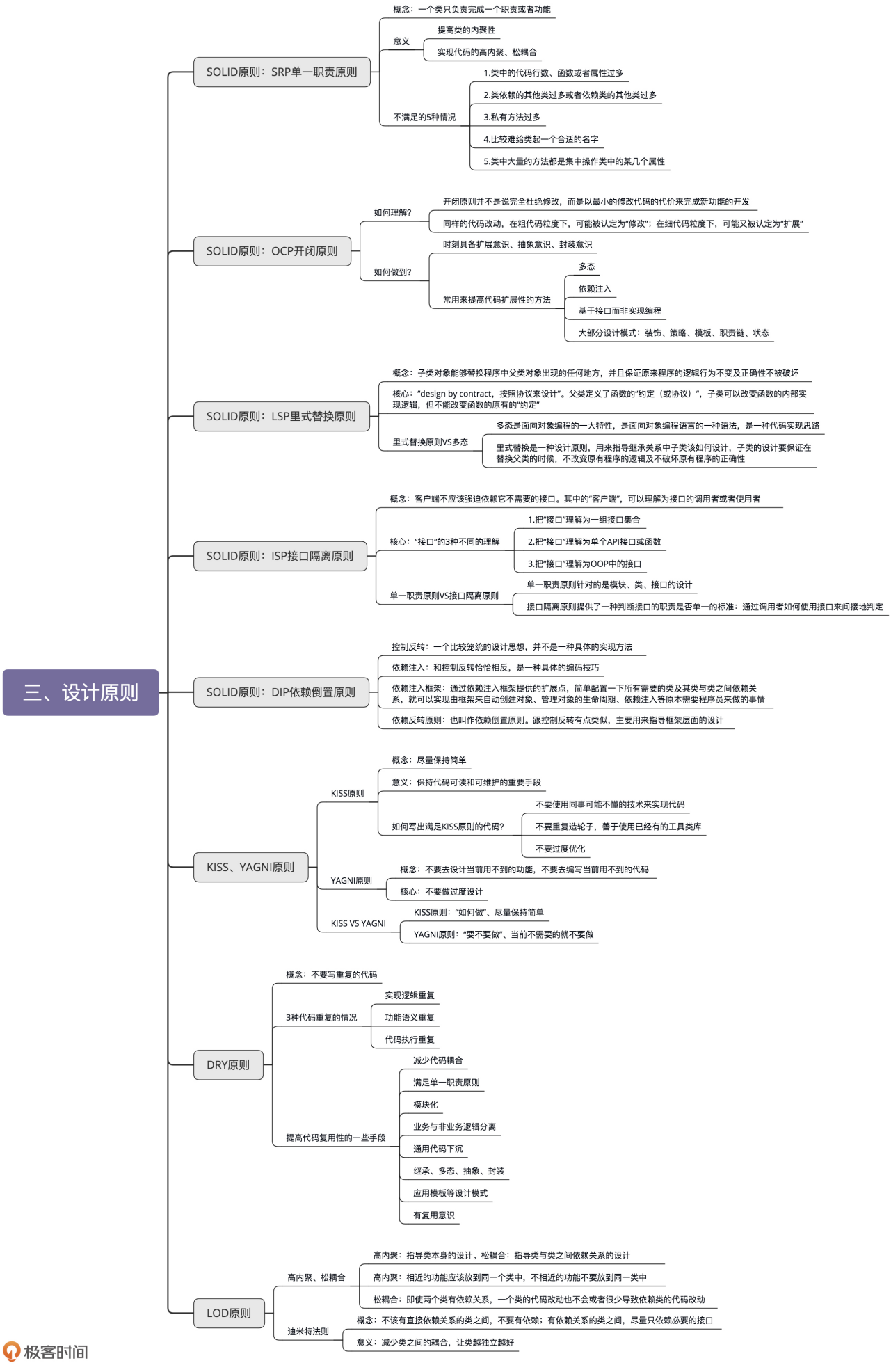
8.LOD 原则

如何理解“高内聚、松耦合”？

“高内聚、松耦合”是一个非常重要的设计思想，能够有效提高代码的可读性和可维护性，缩小功能改动导致的代码改动范围。“高内聚”用来指导类本身的设计，“松耦合”用来指导类与类之间依赖关系的设计。所谓高内聚，就是指相近的功能应该放到同一个类中，不相近的功能不要放到同一类中。相近的功能往往会被同时修改，放到同一个类中，修改会比较集中。所谓“松耦合”指的是，在代码中，类与类之间的依赖关系简单清晰。即使两个类有依赖关系，一个类的代码改动也不会或者很少导致依赖类的代码改动。

如何理解“迪米特法则”？

迪米特法则的描述为：不该有直接依赖关系的类之间，不要有依赖；有依赖关系的类之间，尽量只依赖必要的接口。迪米特法则是希望减少类之间的耦合，让类越独立越好。每个类都应该少了解系统的其他部分。一旦发生变化，需要了解这一变化的类就会比较少。


三、设计原则
SOLID原则：SRP单一职责原则
概念：一个类只负责完成一个职责或者功能
意义
提高类的内聚性
实现代码的高内聚、松耦合
不满足的5种情况
1.类中的代码行数、函数或者属性过多
2.类依赖的其他类过多或者依赖类的其他类过多
3.私有方法过多
4.比较难给类起一个合适的名字
5.类中大量的方法都是集中操作类中的某几个属性
SOLID原则：OCP开闭原则
如何理解?
开闭原则并不是说完全杜绝修改，而是以最小的修改代码的代价来完成新功能的开发
同样的代码改动，在粗代码粒度下，可能被认定为“修改”；在细代码粒度下，可能又被认定为“扩展”
如何做到?
时刻具备扩展意识、抽象意识、封装意识
常用来提高代码扩展性的方法
多态
依赖注入
基于接口而非实现编程
大部分设计模式：装饰、策略、模板、职责链、状态
SOLID原则：LSP里式替换原则
概念：子类对象能够替换程序中父类对象出现的任何地方，并且保证原来程序的逻辑行为不变及正确性不被破坏
核心：“design by contract，按照协议来设计”。父类定义了函数的“约定（或协议）”，子类可以改变函数的内部实现逻辑，但不能改变函数的原有的“约定”
里式替换原则VS多态
多态是面向对象编程的一大特性，是面向对象编程语言的一种语法，是一种代码实现思路
里式替换是一种设计原则，用来指导继承关系中子类该如何设计，子类的设计要保证在替换父类的时候，不改变原有程序的逻辑及不破坏原有程序的正确性
SOLID原则：ISP接口隔离原则
概念：客户端不应该强迫依赖它不需要的接口。其中的“客户端”，可以理解为接口的调用者或者使用者
核心：“接口”的3种不同的理解
1.把“接口”理解为一组接口集合
2.把“接口”理解为单个API接口或函数
3.把“接口”理解为OOP中的接口
单一职责原则VS接口隔离原则
单一职责原则针对的是模块、类、接口的设计
接口隔离原则提供了一种判断接口的职责是否单一的标准：通过调用者如何使用接口来间接地判定
SOLID原则：DIP依赖倒置原则
控制反转：一个比较笼统的设计思想，并不是一种具体的实现方法
依赖注入：和控制反转恰恰相反，是一种具体的编码技巧
依赖注入框架：通过依赖注入框架提供的扩展点，简单配置一下所有需要的类及其类与类之间依赖关系，就可以实现由框架来自动创建对象、管理对象的生命周期、依赖注入等原本需要程序员来做的事情
依赖反转原则：也叫作依赖倒置原则。跟控制反转有点类似，主要用来指导框架层面的设计
KISS、YAGNI原则
KISS原则
概念：尽量保持简单
意义：保持代码可读和可维护的重要手段
如何写出满足KISS原则的代码?
不要使用同事可能不懂的技术来实现代码
不要重复造轮子，善于使用已经有的工具类库
不要过度优化
YAGNI原则
概念：不要去设计当前用不到的功能，不要去编写当前用不到的代码
核心：不要做过度设计
KISS VS YAGNI
KISS原则：“如何做”、尽量保持简单
YAGNI原则：“要不要做”、当前不需要的就不要做
DRY原则
概念：不要写重复的代码
3种代码重复的情况
实现逻辑重复
功能语义重复
代码执行重复
提高代码复用性的一些手段
减少代码耦合
满足单一职责原则
模块化
业务与非业务逻辑分离
通用代码下沉
继承、多态、抽象、封装
应用模板等设计模式
有复用意识
LOD原则
高内聚、松耦合
高内聚：指导类本身的设计。松耦合：指导类与类之间依赖关系的设计
高内聚：相近的功能应该放到同一个类中，不相近的功能不要放到同一类中
松耦合：即使两个类有依赖关系，一个类的代码改动也不会或者很少导致依赖类的代码改动
迪米特法则
概念：不该有直接依赖关系的类之间，不要有依赖；有依赖关系的类之间，尽量只依赖必要的接口
意义：减少类之间的耦合，让类越独立越好

极客时间

四、规范与重构

1. 重构概述

重构的目的：为什么重构（why）？

对于项目来言，重构可以保持代码质量持续处于一个可控状态，不至于腐化到无可救药的地步。对于个人而言，重构非常锻炼一个人的代码能力，并且是一件非常有成就感的事情。它是我们学习的经典设计思想、原则、模式、编程规范等理论知识的练兵场。

重构的对象：重构什么（what）？

按照重构的规模，我们可以将重构大致分为大规模高层次的重构和小规模低层次的重构。大规模高层次重构包括对代码分层、模块化、解耦、梳理类之间的交互关系、抽象复用组件等等。这部分工作利用的更多的是比较抽象、比较顶层的设计思想、原则、模式。小规模低层次的重构包括规范命名、注释、修正函数参数过多、消除超大类、提取重复代码等编程细节问题，主要是针对类、函数级别的重构。小规模低层次的重构更多的是利用编码规范这一理论知识。

重构的时机：什么时候重构（when）？

我反复强调，我们一定要建立持续重构意识，把重构作为开发必不可少的部分融入到开发中，而不是等到代码出现很大问题的时候，再大刀阔斧地重构。

重构的方法：如何重构（how）？

大规模高层次的重构难度比较大，需要有组织、有计划地进行，分阶段地小步快跑，时刻保持代码处于一个可运行的状态。而小规模低层次的重构，因为影响范围小，改动耗时短，所以，只要你愿意并且有时间，随时随地都可以去做。

2. 单元测试

什么是单元测试?

单元测试是代码层面的测试，用于测试“自己”编写的代码的逻辑正确性。单元测试顾名思义是测试一个“单元”，这个“单元”一般是类或函数，而不是模块或者系统。

为什么要写单元测试?

单元测试能有效地发现代码中的 Bug、代码设计上的问题。写单元测试的过程本身就是代码重构的过程。单元测试是对集成测试的有力补充，能帮助我们快速熟悉代码，是 TDD 可落地执行的折中方案。

如何编写单元测试?

写单元测试就是针对代码设计覆盖各种输入、异常、边界条件的测试用例，并将其翻译成代码的过程。我们可以利用一些测试框架来简化测试代码的编写。对于单元测试，我们需要建立以下正确的认知：

编写单元测试尽管繁琐，但并不是太耗时；

我们可以稍微放低单元测试的质量要求；

覆盖率作为衡量单元测试好坏的唯一标准是不合理的；

写单元测试一般不需要了解代码的实现逻辑；

单元测试框架无法测试多半是代码的可测试性不好。

单元测试为何难落地执行?

一方面，写单元测试本身比较繁琐，技术挑战不大，很多程序员不愿意去写。另一方面，国内研发比较偏向“快糙猛”，容易因为开发进度紧，导致单元测试的执行虎头蛇尾，最后，没有建立对单元测试的正确认识，觉得可有可无，单靠督促很难执行得很好。

3. 代码的可测试性

什么是代码的可测试性?

粗略地讲，所谓代码的可测试性，就是针对代码编写单元测试的难易程度。对于一段代码，如果很难为其编写单元测试，或者单元测试写起来很费劲，需要依靠单元测试框架很高级的特性，那往往就意味着代码设计得不够合理，代码的可测试性不好。

编写可测试性代码的最有效手段

依赖注入是编写可测试性代码的最有效手段。通过依赖注入，我们在编写单元测试代码的时候，可以通过 mock 的方法将不可控的依赖变得可控，这也是我们在编写单元测试的过程中最有技术挑战的地方。除了 mock 方式，我们还可以利用二次封装来解决某些代码行为不可控的情况。

常见的 Anti-Patterns

典型的、常见的测试不友好的代码有下面这 5 种：

代码中包含未决行为逻辑；

滥用可变全局变量；

滥用静态方法；

使用复杂的继承关系；

高度耦合的代码。

4. 大型重构：解耦

“解耦”为何如此重要?

过于复杂的代码往往在可读性、可维护性上都不友好。解耦，保证代码松耦合、高内聚，是控制代码复杂度的有效手段。如果代码高内聚、松耦合，也就是意味着，代码结构清晰、分层、模块化合理、依赖关系简单、模块或类之间的耦合小，那代码整体的质量就不会差。

代码是否需要“解耦”？

间接的衡量标准有很多，比如：改动一个模块或类的代码受影响的模块或类是否有很多、改动一个模块或者类的代码依赖的模块或者类是否需要改动、代码的可测试性是否好等等。直接的衡量标准是把模块与模块之间及其类与类之间的依赖关系画出来，根据依赖关系图的复杂性来判断是否需要解耦重构。

如何给代码“解耦”？

给代码解耦的方法有：封装与抽象、中间层、模块化，以及一些其他的设计思想与原则，比如：单一职责原则、基于接口而非实现编程、依赖注入、多用组合少用继承、迪米特法则。

当然，还有一些设计模式，比如观察者模式。

5. 小型重构：编码规范

前面我们讲了很多设计原则，后面还会讲到很多设计模式，利用好它们都可以有效地改善代码的质量。但是，这些知识的合理应用非常依赖个人经验，有时候用不好会适得其反。但是编码规范正好相反，大部分都简单明了，在代码的细节方面，能立竿见影地改善质量。除此之外，我们前面也讲到，持续低层次小规模重构依赖的基本上都是这些编码规范，也是改善代码可读性的有效手段。

根据我自己的开发经验，我总结罗列了 20 条我认为最应该关注、最好用的编码规范，分为三个大的方面：命名与注释（Naming and Comments）、代码风格（Code Style）、编程技巧（Coding Tips）。

命名与注释

命名的关键是能准确的达意。对于不同作用域的命名，我们可以适当的选择不同的长度，作用域小的命名，比如临时变量等，可以适当的选择短一些的命名方式。除此之外，命名中个也可以使用一些耳熟能详的缩写。

我们借助类的信息来简化属性、函数的命名，利用函数的信息来简化函数参数的命名。

命名要可读、可搜索。不要使用生僻的、不好读的英文单词来命名。除此之外，命名要符合项目的统一规范，也不要用些反直觉的命名。

接口有两种命名方式。一种是在接口中带前缀"I"，另一种是在接口的实现类中带后缀“Impl”。两种命名方式都可以，关键是要在项目中统一。对于抽象类的命名，我们更倾向于带有前缀“Abstract”。

注释的目的就是让代码更容易看懂，只要符合这个要求，你就可以写。总结一下的话，注释主要包含这样三个方面的内容：做什么、为什么、怎么做。对于一些复杂的类和接口，我们可能还需要写明“如何用”。

注释本身有一定的维护成本，所以并非越多越好。类和函数一定要写注释，而且要写的尽可能全面详细些，而函数内部的注释会相对少一些，一般都是靠好的命名和提炼函数、解释性变量、总结性注释来做到代码易读。

代码风格

代码风格都没有对错和优劣之分，不同的编程语言风格都不太一样，只要能在团队、项目中统一即可，不过，最好能跟业内推荐的风格、开源项目的代码风格相一致。所以，这里就不展开罗列了，你可以对照着自己熟悉的编程语言的代码风格，自己复习一下。

编程技巧

将复杂的逻辑提炼拆分成函数和类；

通过拆分成多个函数的方式来处理参数过多的情况；

通过将参数封装为对象来处理参数过多的情况；

函数中不要使用参数来做代码执行逻辑的控制；

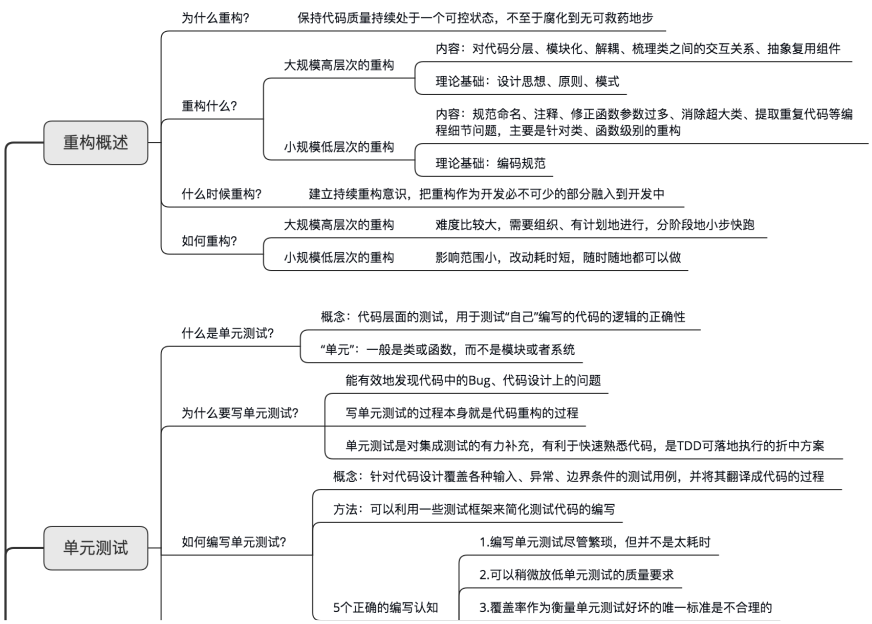
移除过深的嵌套层次，方法包括：去掉多余的 if 或 else 语句，使用 continue、break、return 关键字提前退出嵌套，调整执行顺序来减少嵌套，将部分嵌套逻辑抽象成函数；

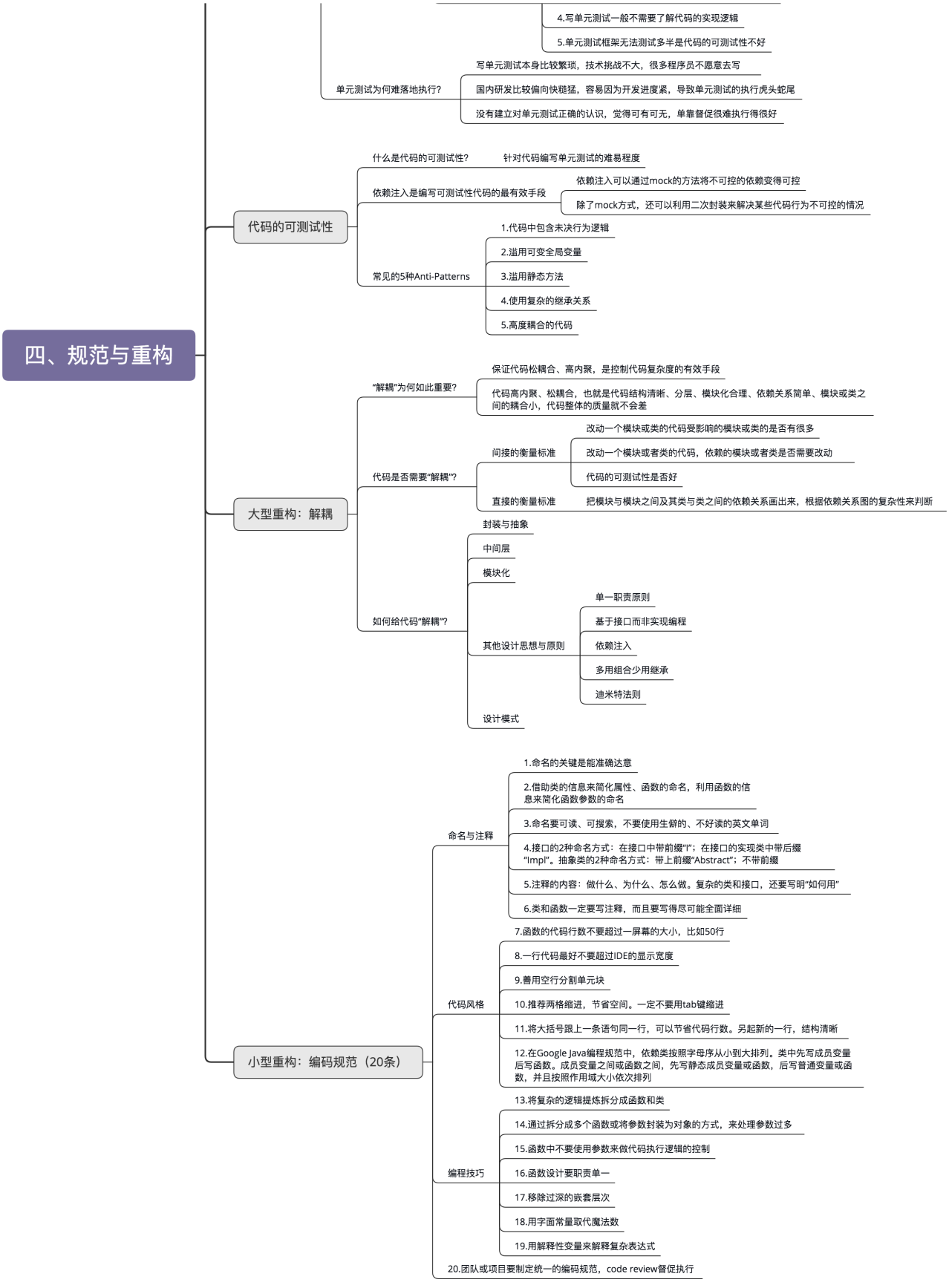
用字面常量取代魔法数；

利用解释性变量来解释复杂表达式。

统一编码规范

除了细节的知识点之外，最后，还有一条非常重要的，那就是，项目、团队，甚至公司，一定要制定统一的编码规范，并且通过 Code Review 督促执行，这对提高代码质量有立竿见影的效果。





课堂讨论

不知不觉我们已经学了这么多内容，在复习完这部分内容之后，你可以在留言区说一说你的掌握程度，看自己符合我开篇中讲到的哪个层次。

如果有帮助，欢迎你收藏这篇文章，并且把它分享给你的朋友。

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇　37 | 实战二（下）：重构ID生成器项目中各函数的异常处理代码

下一篇　39 | 运用学过的设计原则和思想完善之前讲的性能计数器项目（上）

精选留言 (17)

写留言

南山
2020-01-29

这篇看起来反而是最轻松的。
一直有每篇都用xmind做笔记的习惯，自问自答，wwh。
会结合工作中的代码去思考内容，审视代码，也在code review中和同事结合团队现状制定可行的checklist和编码规范
受益良多，也更期待后续

展开

6

志振爱咋滴咋滴
2020-02-01

蓦然回首，发现原来已经学习了那么多的知识了，其实很多东西都是在日常工作中需要不断注意，思考，审视的内容，希望在以后的工作里，能够时刻提醒自己，注意代码质量，提高重构意识

守拙
2020-01-30

对于面向对象和设计原则的掌握程度要稍好一些, 对于单测,重构等章节学的不太认真.
感谢作者精心安排的课程,非常细,有作者自己的理解而不是洗书.
希望后面的具体设计模式会更精彩!

展开

Geek_27a248
2020-01-30

感觉差的好多，我感觉我现在还处在理解概念阶段，写代码时不能灵活使用

webmin
2020-01-30

需要长期练习，有切身体验能记住，没有的还是还给老师了。

辣么大
2020-01-29

1. 学完了还是得用：自己业余时间写了一个小项目，然后尝试丰富功能，重构它。目前完成了最基本的功能，代码写的只能算是能用。基本每天都在使用这个小工具。下步打算利用课程中的知识来“盘他”！
2. 自己个人代码中加入了 FindBugs 和 Google code style。
3. Leetcode练习时注意逐步优化，想想也算是重构吧，不用ide写代码，注意编码规范。...

展开

javaadu
2020-01-29

想了几次，抽象并不是面向对象设计独有的特性，而是一个最基本的应对软件复杂性的思路，大型软件如果想活得好，抽象思维一定少不了。

之前教科书中都说面向对象是三大特性：封装、继承、多态，这里不知道老师是怎么看的

展开

刘大明
2020-01-29

今天学习了总结面向对象，设计原则，编码规范和重构相关的知识总结。
面向对象只是有一些概念更加熟悉了。
设计原则我觉得自己理解了。
编码规范和重构是我工作中一直在用的，这次看了专栏之后有了很多新的认识。
前面的课程要自己抽时间做个思维导图，争哥的导图做的很好，已经下载收藏。

展开

xavier
2020-01-29

基本是之前每篇的总结，读起来也较轻松，接下来需自己再总结一下。

Jxin
2020-01-29

1.额。。刚好和这几天对这个专栏整理的xmind差不多。也建议大家自己写下xmind导图，对加深印象和理解很有帮助。串起来和零散的知识点，可记忆性和可理解性差很多。

2.很好奇怎么讲这些设计模式，期待再次被惊艳。我自己是扒spring对各种设计模式的应用后，才get到在什么场景用什么模式有什么好处这些点的。讲道理，单知道设计模式怎…

展开

javaadu
2020-01-29

不同模块的掌握程度不一样：
1. 设计原则那块有些原则是只是有个印象，有些则是在项目中应用，整体上看这块还需要更多的复习和理解
2. 重构和编程规范这块在工作中一直持续在做，还不错
3. 面向对象部分也是有不错的知识结构，并且在工作中的项目里实践…

展开

Yang
2020-01-29

有些具体的设计思想、原则概念性的问题有点含糊不清，不过现在写代码时会考虑代码的可读性、可维护、可扩展性等。真的很感谢老师出了这门课，之前每次写代码都纠结代码质量怎么保证，老师的这门课解答了我之前所有关于代码质量的疑问。

展开

六号先生117

2020-01-29

老师真好，昨天正好开始复习，自己画思维导图，看到文中总结结构图真棒

Eden Ma

2020-01-29

34分钟的音频,给力.不知不觉已经学习三个月了~

展开

Algo

2020-01-29

打卡！ 前几天有自己将SOLID的内容总结一遍。今天更系统复习了第一遍。只是实际开发中怎样不会顾此失彼，全靠小跑重构么。

展开

ちよくん

2020-01-29

打卡，我是理解记忆，老师的原话我肯定记不住，我在老师理解的基础上有自己的理解！

Jeff.Smile

2020-01-29

今天大年初五，抢占沙发！

展开